Page/slab cache control in a virtualized environment

Balbir Singh
Linux Technology Center, IBM,
balbir@linux.vnet.ibm.com

Abstract

The Linux page/slab cache subsystems are one of the most useful subsystems in the Linux kernel. Any attempts to limit its usage have been discouraged and frowned upon in the past. However, virtualization is changing the role of the kernel running on the system, specifically when the kernel is running as a guest. Assumptions about using all available memory as cache and optimizations will need need to be re-looked in an environment where resources are not fully owned by one guest OS.

In this paper, we discuss some of the pain points of page cache in a virtualized environment; like double caching of data in both the host and guest and its impact on memory utilization. We look at the current page cache behavior of Linux running as a guest and when multiple instances of guest operating systems are running. We look at current practices and propose new solutions to the solving the double caching problem in the kernel.

1 Introduction

The cache systems are typically designed to grow, they tend to use as much memory is required for caching key data that can be reused later. They also provide a reclaim system that can quickly reclaim back memory used for caching. An often asked question on the Linux Kernel Mailing List (LKML) [5] relates to why the free memory on the system is very low, even though the system is mostly idle or even when the system has few applications that do not take up a lot of memory. Typically we distinguish between free memory and freeable memory. The cache (unless dirty) falls in the category of freeable memory. We use memory to optimize the cost of otherwise reading from a slow device. The ability to reclaim from the cache when needed is a good design trade-off.

The scenario is quite different in a virtualized environment. The entire guest kernel memory is mapped into the hypervisor address space. The memory cached in the kernel, shows up as mapped memory in the hypervisor. Beyond the change of the way memory is visible, caching policies in both the guest and host can lead to double caching. Double caching is not very good in a virtualized environment, where resources are scarce and heavily shared. In the sections to follow, we look at page cache in a virtualized environment, some basic data about page cache in a virtualized environment, our approaches to solving the problem, future work and we finally conclude with recommendations.

NOTE: We've used the terms host and hypervisor interchangeably in this paper.

2 I/O in a Virtualized Environment

Our focus in this paper is on the KVM hypervisor [3] and the Linux Operating System running as the guest operating system. The KVM hypervisor configuration can be very complex. Lets look at the various ways of carrying out I/O.

1. **Direct Assignment**: In this mode, the IOMMU [2] creates one or more unique address spaces which can be used for DMA operations. With IOMMU's and direct assignment, a device can be assigned to a virtual machine directly. This speeds up I/O immensely. The drawback of such a scheme is scalability. There are standards that allow to solve the scalability problem by virtualizing the workqueues, interrupts, registers on a per VM basis while using the same device. The guest drivers need to support these devices to make full use of the capabilities.

2. **Paravirtualized I/O**: The hypervisor uses the virt I/O [6] subsystem to paravirtualize the I/O and makes as efficient as possible. The data exchanged

between the guest and the host is done via a zero-copy mechanism, with efficient notification mechanism for availability of data. This mode requires support from the guest operating system to have paravirtualized drivers.

3. **Emulated I/O**: In this mode, the hypervisor emulates a storage device. Guest drivers do I/O to the emulated device and the emulated device in-turn does I/O to the actual physical device.

Modes 2 and 3 above need support from the hypervisor to carry out the complete I/O. Beyond the I/O modes listed above, virtual machines themselves can be configured in

1. **Dedicated Partition Mode**: In this mode, the virtual machine is installed the file system on a partition. This could be an entire disk, a virtual partition spanning multiple disks, an LVM partition or a disk partition.
2. **Virtual Machine Image Mode**: In this mode, the virtual machine is installed in an image file. A set of Virtual Machine Images (VMI) are kept together in a virtual machine repository

Understanding the details of the various image formats is essential to identify the cost of doing I/O operations and hence the levels of caching and the cost of caching data in memory. In this paper, we don't focus on any specific image file format. The focus is on common strategies.

3 Page Caching Strategies

There are various strategies that one can employ for page cache. The strategies are examined here

3.1 Guest Only Caching

In the guest only caching strategy, the host page cache is bypassed. This is done by passing the `cache=none` argument to the hypervisor during guest startup. This option enables direct I/O and directly writes the data to disk, bypassing the host page cache. This strategy works well for cases where the filesystem of the VM is dedicated to the guest using direct assignment for I/O or if the guest works in dedicated partition mode. Guest only caching mode can be used with VMI's, but it can be an ineffective strategy if the hypervisor is doing I/O. Several VM's running in parallel, have their own I/O scheduler and if the host does not merge the I/O's, it can cause excessive head movement in seeking devices. If there are several VM's running in parallel, they could cache memory proportional to their size. The total consumption of memory in each of the guests for caching can be very high. This consumption shows up as mapped memory in the hypervisor. The most effective way to free the memory cached in the guests is through ballooning. This requires that we have an auto ballooning daemon running in the background and a cooperative guest. [1]

3.2 Host Only Caching

In the host only caching strategy, the guest cache is not used for caching. All the caching is delegated to the host. This works well for VMI's, the host page cache optimizes disk I/O. The host is able to optimize I/O from all VM's and provides higher throughput. KVM supports `writethrough` and `writeback` caching. In writethrough caching, the I/O is blocked till the data hits the disk. In writeback mode, the I/O returns as soon as the data hits the host page cache. The big advantage of the writeback mode is the throughput, the biggest disadvantage is potential of data loss if the hypervisor crashes. True host only caching is not possible, each guest maintains its own cache, which leads to mixed caching. Typical recommendations to reduce guest caching include changing the setting of `vm.swappiness` to 0. In section 4 we look at the results from the various modes mentioned in this section, including results when `vm.swappiness` is set to 0.

3.3 Mixed Caching

In this mode, both the host and the guests cache I/O data. This happens in a typical VM setup. The disadvantages listed in section 3.2 apply to this strategy. Beyond that a system with caching both on the host and the guest(s) incurs a penalty of double memory usage for caching the same data. While there are several ways to deal with page duplication problem [1], none of them deal with the duplication of page cache between the host and the guest.

[1] A guest is considered cooperative if it has a balloon driver enabled and running

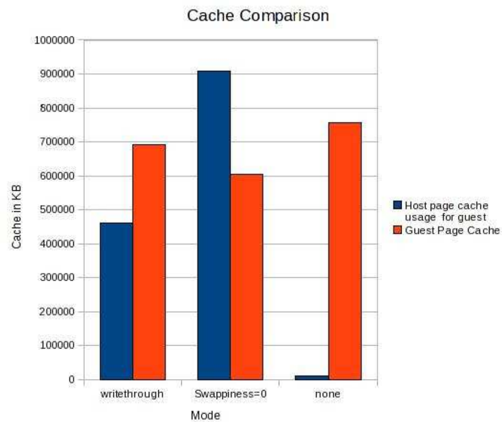

Figure 1: Cache Usage in various modes

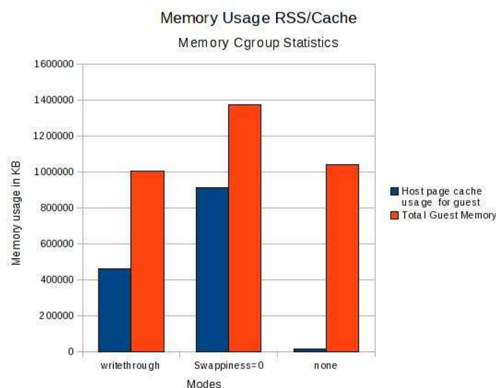

Figure 2: Host Page Cache and Guest RSS Usage in various modes

4 Page Cache Control

The double caching behaviour of was studied using memory cgroups [7]. A new cgroup was created for the virtual machine being executed. The virtual machine(s) ran the kernbench [8] benchmark. Memory cgroups can provide information about the RSS and page cache (mapped and unmapped) usage of the process running inside the cgroup (in this case the virtual machine comprises of the processes running as a part of hypervisor). Each VM was allocated 1 gigabyte of RAM and 2 Virtual CPUs (VCPUs)

Figure 1 shows the unmapped cache usage in three modes.

1. The first mode is the writethrough mode, which was described earlier in section 3.2. The guest and host both consume memory for page cache simultaneously and independently. The guest usage is however larger than the host usage. The data showed 60% of the data was duplicated over the entire run of the benchmark.

2. The second mode is the writeback with swappiness in the guest set to 0. The results showed that the guest page cache usage was lower than the host page cache usage and also lower than the usage in writethrough mode. The usage however was not close to 0, it was close to 50% of the host page cache usage. The host page cache usage was quite high.

3. The last mode is the direct I/O or the `cache=none` mode. In this mode, the hypervisor uses direct I/O to write out the pages from the guest block device to the disk. The data shows that the host page cache usage for the virtual machine is almost 0, all the caching is done in the guest. The size of the cache in the guest is high and higher than the other modes experimented with.

Figure 2 shows the page cache usage on behalf of the guest versus the RSS of the virtual machine. The results show that in addition to the memory being occupied by as cache in the guest (which shows up under RSS usage in the figure), the host is also caching page cache data. The key observations are

1. Host side caching for `cache=none` is almost 0 as expected.

2. With `cache=writethrough`, there is still double caching. The host uses close to 40% of the guest memory for caching data

3. When swappiness is set to 0 and the mode is `cache=writeback`, the host uses additional memory to cache guest data.

5 Proposed Approach

The proposed approach consists of two mechanisms to reduce the double caching of page cache data. The approaches are discussed

5.1 Mixed Caching With Host Emphasis

In this mechanism, both the guest and host use memory for page cache, but the cache is primarily pushed towards the host page cache. The guest page cache is monitored and shrunk frequently. The kernel has a partial implementation of this approach for NUMA systems [4] when the `zone_reclaim_distance` is greater than 0, implying that the cost of allocation from different nodes is high, the code does local reclaim of easy to free pages before allocating from a distant node. The algorithm reuses this behaviour and exploits the `min_unmapped_ratio` to keep the unmapped page cache usage under control.

Algorithm 1 Modified VM algorithm for page cache control

```
get_page_from_freelist()
...
determine zone to allocate from
if zone is below watermark then
  if should_balance_unmapped_cache() then
    wakeup kswapd
  end if
end if
```

Algorithm 2 Check if page cache should be controlled

```
should_balance_unmapped_cache()
if unmapped pages for zone > min_unmapped_ratio * number of zone pages then
  return TRUE
else
  return FALSE
end if
```

Algorithm 3 Kswapd changes

```
balance_pgdat()
...
on wakeup check if zone is below watermark or should_balance_unmapped_cache()
if unmapped pages need balancing then
  Reuse zone_reclaim logic
  for various reclaim priorities do
    Invoke reclaim targeting only unmapped pages and with swapping out of pages disabled
  end for
end if
```

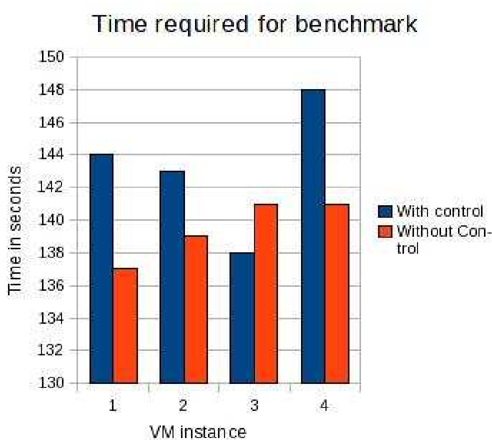

Figure 3: Time comparison of kernbench for with and without changes

Algorithms 1, 2, 3 show the changes made to control unmapped pages in the page cache. The code provides control over unmapped page cache via a boot parameter called `unmapped_page_control`. This boot parameter selectively activates the page cache control feature. By default 1% of the memory can be used for unmapped page cache. There is a sysctl `vm.min_unmapped_ratio` that can be tuned in the guest to control the amount of unmapped page cache.

5.1.1 Experiments and Results

The approach was tested by running four VM's in parallel, each running kernbench. Each VM had 1 GB of memory and 2 VCPUs.

The figure shows an overhead of close to 5% when the feature is enabled with `min_unmapped_ratio` set to 1%.

Figure 4 shows the free and cached memory usage of the benchmark running in four VM's without any changes to support control of unmapped pages. As can be seen, the free memory is low and the unmapped page cache memory usage is high. Figure 5 shows the free and cached memory usage of the same benchmark running in four VM's with the `unmapped_page_control` boot parameter specified during bootup. The figure shows a higher free memory and lower unmapped page cache utilization.

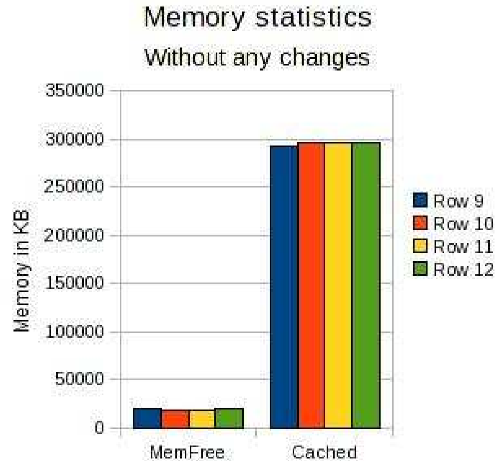

Figure 4: Free and cached memory inside the guest without any changes

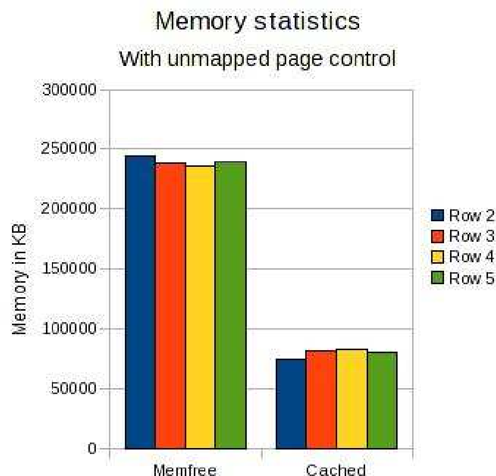

Figure 5: Free and cached memory inside the guest with changes

5.2 Cooperative Unmapped Page Cache Control

In this mechanism, the ballooning driver is used to cooperatively control page cache. In contrast to the previous approach, this approach is activated selectively on memory pressure within the hypervisor. The code changes in this approach are quite simple and consists of the following:

1. Create a new GFP flag, called `__GFP_FREE_CACHE`
2. Use `__GFP_FREE_CACHE` from the balloon driver, when it allocates pages under pressure.
3. The virtual memory subsystem honours the `__GFP_FREE_CACHE` flags by reusing code from `zone_reclaim` and the approach above to free both unmapped page cache and slab cache pages when the guest operating system is ballooned.

The key challenge with this approach is that the cache usage is externally controlled when ballooning occurs. It is important to make the correct decisions on when to balloon a particular guest and by how much. Typically a hypervisor would have a automatic tuning daemon whose job is to monitor memory usage in the host, the free memory, memory pressure in the host, the guest memory usage, various entitlements and makes smart decisions on which guests to balloon [2]. For the experiments and results obtained using this approach, we used a similar tool to monitor and automatically balloon the guests as required.

5.2.1 Experiments and Results

The test setup involved four VM's all running kernbench with 4 VCPUs and 6GB of memory. Four guest VM's ran this test in parallel. The test was run under a memory monitor as described in section 5.2, which means it was subjected to auto ballooning based on host memory pressure, the size, usage and entitlement of each of the VM's. As stated earlier, the ballooning operation could increase or decrease the memory footprint of the guest VM.

[2] A ballooning operation can either reduce the memory footprint of the guest or give it additional memory to use

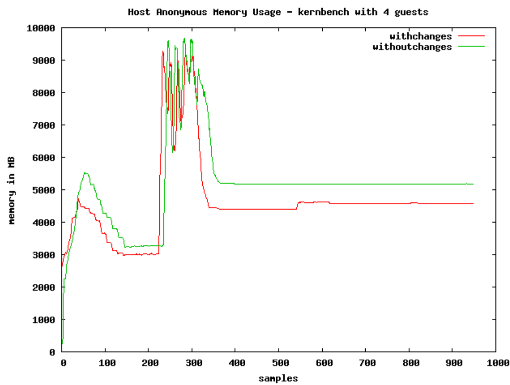

Figure 6: Host Anonymous Memory, running kernbench four VM's

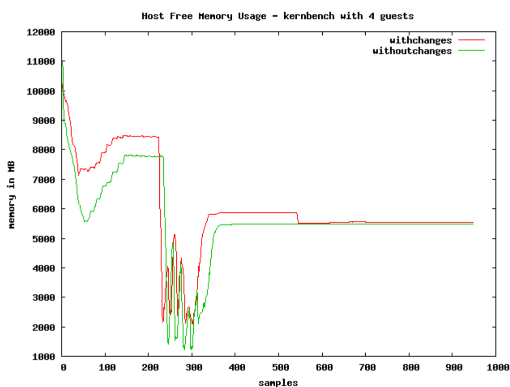

Figure 7: Host Free Memory, running kernbench four VM's

	VM	With Changes	Without Changes
make -j3	1	88.83	87.582
make -j16	1	76.786	76.686
make -j3	2	88.124	87.463
make -j16	2	77.264	76.704
make -j3	3	88.808	87.544
make -j16	3	76.748	75.522
make -j3	4	88.128	87.436
make -j16	4	76.828	75.63

Table 1: Elapsed time four VM's running kernbench

Figure 6 shows the anonymous memory usage in the host and correspondingly figure 7 shows the in the graph show the usage and free memory with and without changes to the operating system for cooperative ballooning. Figure 6 shows that the anonymous memory usage after the changes is lower as expected. This indicates that the cooperative ballooning technique, reduces the cache size and in turn the RSS size of each guest VM[3]. Similarly figure 7 shows that the free memory in the host is higher with changes.

Table 1 shows the results for the kernbench run in each VM with and without the ballooning changes for cooperative page cache management. The results show no significant overhead of the patches, but as the graphs earlier depict, it results in higher free memory in the hypervisor.

6 Future Work

The approaches listed in the paper are by no means complete. There are several additional possibilities to reduce page cache deduplication. One of them is to extend KSM [1] to deal with page cache data between host and guest operating systems. There is also additional scope in paravirtualizing hints such as `madvise(2)`, so that the hypervisor is aware of the hints and can appropriately handle the hints and manipulate its page cache usage in line with the hints coming from the applications running in the guest operating system.

7 Conclusion

Our results show that there is definitely double caching of page cache data between the hypervisor and guests.

[3] As seen from the host

Our approach pushes the caching of page cache data (more specifically unmapped page cache) to the hypervisor. The approaches listed above *unmapped page control* and *cooperative page cache control*.

The unmapped page control approach provides the best control over double caching and it also provides the flexibility to the user on what percentage of memory can be used for unmapped pages. This approach however, shows a noticeable overhead on the run time, due to the control introduced. We noticed in our experiments that the overheads came from the time required to scan and remove unmapped cached pages, rather than the lack of memory for caching.

In the cooperative approach provides noticeable benefit in terms of free memory available when the technique is used. The technique however, requires a daemon that continuously monitors all the guest operating systems and invokes ballooning operations when necessary. The really good aspect of this approach was minimal to no overhead in implementing this feature.

We believe that both the approaches listed above have an important role to play. The invocation and usage of these approaches is best left to the system administrator/user or a higher level software making decisions for virtualization environments. The key advantage these approaches provide is that they allow more free memory in the hypervisor, which allows additional work to be executed in the hypervisor.

8 Acknowledgements

The author would like to thank the following people for their help and support throughout this effort. Their names appear in no particular order below. Naren Devaiah, Premalatha Nair, Dipankar Sarma, Vaidyanathan Srinivasan, Adam Litke, Joel Schopp, Mike Day, Paul Mckenney, Karl Rister, Avi Kivity, Dave Hansen, Tim Pepper, Anthony Liguory, Rayan Harper, Rik van Riel, Larry Kessler, Ankita Garg, Supriya Kannery, Venkata R Jagana and many others who've been helped me via discussion/suggestions.

9 Legal Statement

©International Business Machines Corporation 2010. Permission to redistribute in accordance with Linux Symposium submission guidelines is granted; all other rights reserved.

This work represents the view of the authors and does not necessarily represent the view of IBM.

IBM, IBM logo, ibm.com, and WebSphere, are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you. This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

References

[1] Andrea Arcangeli, Izik Eidusa, and Chris Wright. Increasing memory density using ksm. In *OLS '09: The 2009 Linux Symposium*, pages 19–28, 2009.

[2] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert van Doorn. The price of safety: Evaluating iommu performance. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 9–20, July 2007.

[3] Avi Kivity. kvm: the linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[4] Christoph Lameter. Local and remote memory. In *Memory in a Linux/NUMA System*, pages 1–25, July 2006.

[5] Linux Kernel Mailing List. http://lkml.org, Last viewed in May 2010.

[6] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[7] Balbir Singh and Vaidayanathan Srinivasan. Containers: Challenges with memory resource controller and its performance. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 209–222, 2007.

[8] Kernbench version 0.42. http://www.kernel.org/pub/linux/kernel/people/ck/apps/kernbench/, Last viewed in May 2010.

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering

Programme Committee

Andrew J. Hutton, *Linux Symposium*
Martin Bligh, *Google*
James Bottomley, *Novell*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Matthew Wilson

Proceedings Committee

Robyn Bergeron

With thanks to
John W. Lockhart, *Red Hat*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.